

First Steps

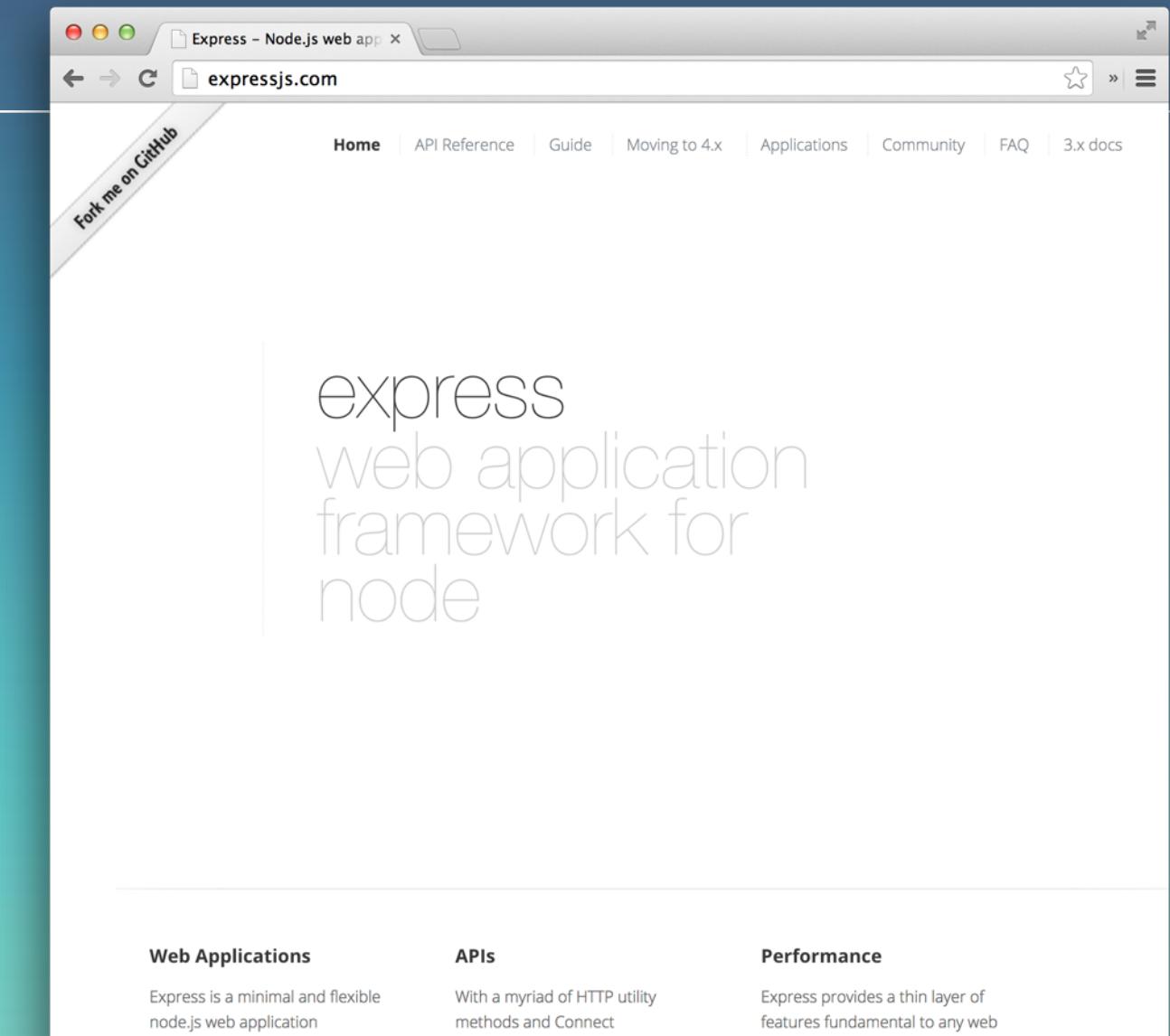
Level 1



What Express Is

A web application framework
for Node

- Minimal and flexible
- Great for building **Web APIs**
- Popular services built on Express
i.e. MySpace, Ghost and more
- Foundation for other tools and
frameworks, like **Kraken** and **Sails**



Installing Express

Node Package Manager

Use **npm** to install the latest stable version

```
$ npm install express
```



Use **@** to install a specific version

```
$ npm install express@4.9
```

```
$ npm install express@3.15.2
```

installs latest version
from the 4.9 branch

installs specific version

This course covers **version 4.9.x**

Code seen here should run on any version of Express
which starts with 4.9 (i.e. 4.9.1, 4.9.2, 4.9.3, etc.)



Writing Hello World

Calling the `express` function gives us an application instance

application instance

app.js

```
var express = require('express');
var app = express();
```



Writing Hello World

The `app.get` function creates a route that accepts GET requests

app.js

```
var express = require('express');
var app = express();

app.get('/', function(request, response) {
  response.send('Hello world');
});

app.listen(3000);
```

binds application
to tcp port 3000

sends back server response

built-in functions
named after HTTP verbs

app.post(...)
app.put(...)
app.patch(...)
app.delete(...)

...



Writing Hello World

The `app.listen` function takes an optional callback, which is called once the app is ready to start taking requests

app.js

```
var express = require('express');
var app = express();

app.get('/', function(request, response) {
  response.send('Hello world');
});

app.listen(3000, function() {
  console.log('Listening on port 3000');
});
```

printed to the console



Running our Express app

Start the server with the node command

```
$ node app.js  
Listening on port 3000
```

Changes to code require a server restart.

Requests with curl

```
$ curl http://localhost:3000/
```

Hello world

server response

Control + C stops the server

```
$ node app.js  
Listening on port 3000
```

^C

interrupts current process

The Request and Response objects

Express extends Node HTTP objects

```
app.get('/', function(request, response) {  
  ...  
});
```

Express
source code

lib/request.js

```
var req = exports = module.exports = {  
  __proto__: http.IncomingMessage.prototype  
};  
...
```

inheritance in
JavaScript

app.js

<https://github.com/strongloop/express>

objects from
Node HTTP

lib/response.js

```
var res = module.exports = {  
  __proto__: http.ServerResponse.prototype  
};  
...
```

Calling Node's HTTP functions

We can respond from Express using Node's `write` and `end` functions

app.js

```
var express = require('express');
var app = express();
```

```
app.get('/', function(request, response) {
  response.write('Hello world');
  response.end();
});
```

using Node API

```
app.listen(3000);
```

Response from both

```
$ curl http://localhost:3000/
```

```
Hello world
```

...very useful when we start writing “extensions” for Express

same thing

```
response.send('Hello world')
```

using Express API



Responding with JSON

The send function converts Objects and Arrays to JSON

app.js

```
app.get('/blocks', function(request, response) {  
  var blocks = ['Fixed', 'Movable', 'Rotating'];  
  response.send(blocks);  
});
```

use -i to print response headers

```
$ curl -i http://localhost:3000/blocks
```

HTTP/1.1 200 OK

X-Powered-By: Express

Content-Type: application/json; charset=utf-8

```
["Fixed", "Movable", "Rotating"]
```

sets proper
response headers



Using the response.json function

The json function reads better when all we respond with is JSON

app.js

```
app.get('/blocks', function(request, response) {  
  var blocks = ['Fixed', 'Movable', 'Rotating'];  
  response.json(blocks);  
});
```

Same response as send, for Objects and Arrays

```
$ curl -i http://localhost:3000/blocks
```

```
HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: application/json; charset=utf-8
```

```
["Fixed", "Movable", "Rotating"]
```



Responding with HTML

The `send` function converts strings to HTML

app.js

```
app.get('/blocks', function(request, response) {
  var blocks = '<ul><li>Fixed</li><li>Movable</li></ul>';
  response.send(blocks);
});
```

Responds with `text/html`

```
$ curl -i http://localhost:3000/blocks
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
```

```
<ul><li>Fixed</li><li>Movable</li></ul>
```

For server-side templates,
checkout EJS or Jade



Redirecting to relative path

The `redirect` function sets the proper response headers

```
app.get('/blocks', function(request, response) {  
  response.redirect('/parts');  
});
```

```
$ curl -i http://localhost:3000/blocks  
  
HTTP/1.1 302 Moved Temporarily  
X-Powered-By: Express  
Location: /parts  
Content-Type: text/plain; charset=utf-8  
  
Moved Temporarily. Redirecting to /parts
```



Redirecting with custom status code

The **status code** can be passed as the first argument to redirect

app.js

```
app.get('/blocks', function(request, response) {  
  response.redirect(301, '/parts');  
});
```



optional status code

```
$ curl -i http://localhost:3000/blocks
```

```
HTTP/1.1 301 Moved Permanently
```

```
X-Powered-By: Express
```

```
Location: /parts
```

```
Content-Type: text/plain; charset=utf-8
```

```
Moved Permanently. Redirecting to /parts
```



Middleware

How They Work

Level 2 - Part I



Rich JavaScript Applications

They allow for a more **interactive** experience on the web.
Let's build this using Express!



Writing index.html

Place HTML files under the **public** folder



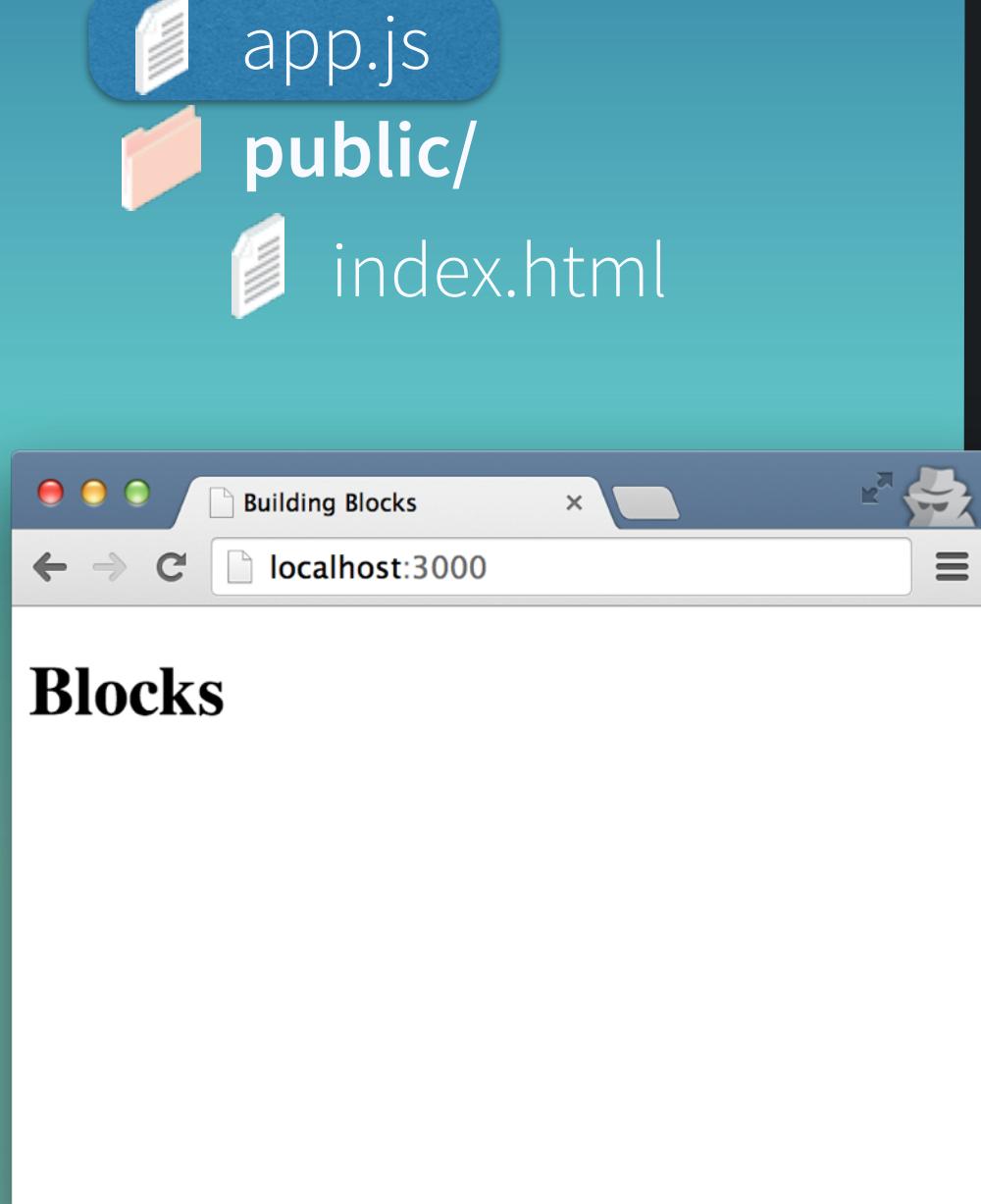
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Building Blocks</title>
</head>
<body>
  <h1>Blocks</h1>
</body>
</html>
```

index.html



Serving files with sendFile

The index.html file is served from Express



```
app.js
```

```
var express = require('express');
var app = express();

app.get('/', function(request, response) {
  response.sendFile(__dirname + '/public/index.html');
});

app.listen(3000);
```

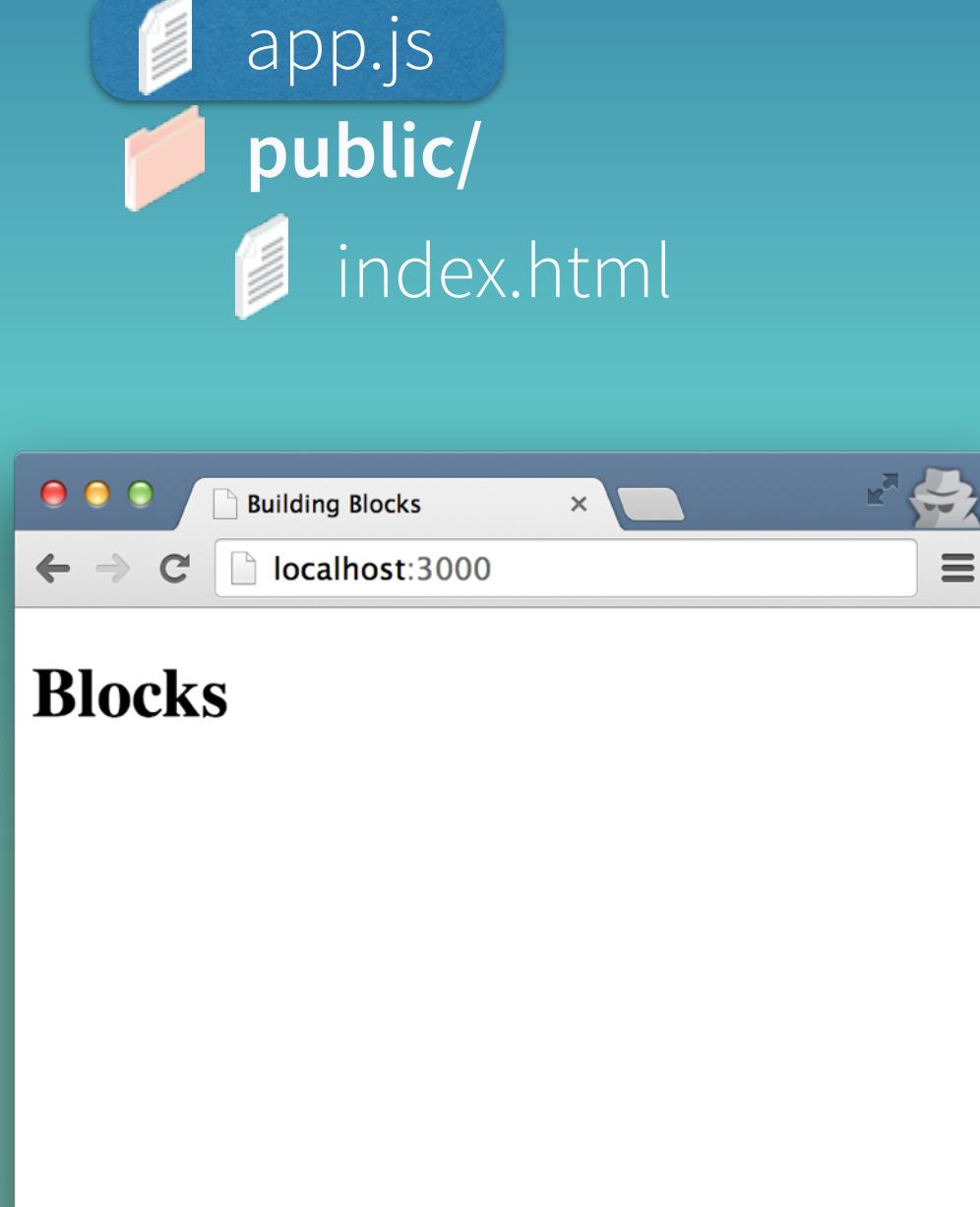


name of the directory the currently
executing script resides in



Mounting middleware

The `app.use` function adds middleware to the application stack



```
app.js
```

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.listen(3000);
```

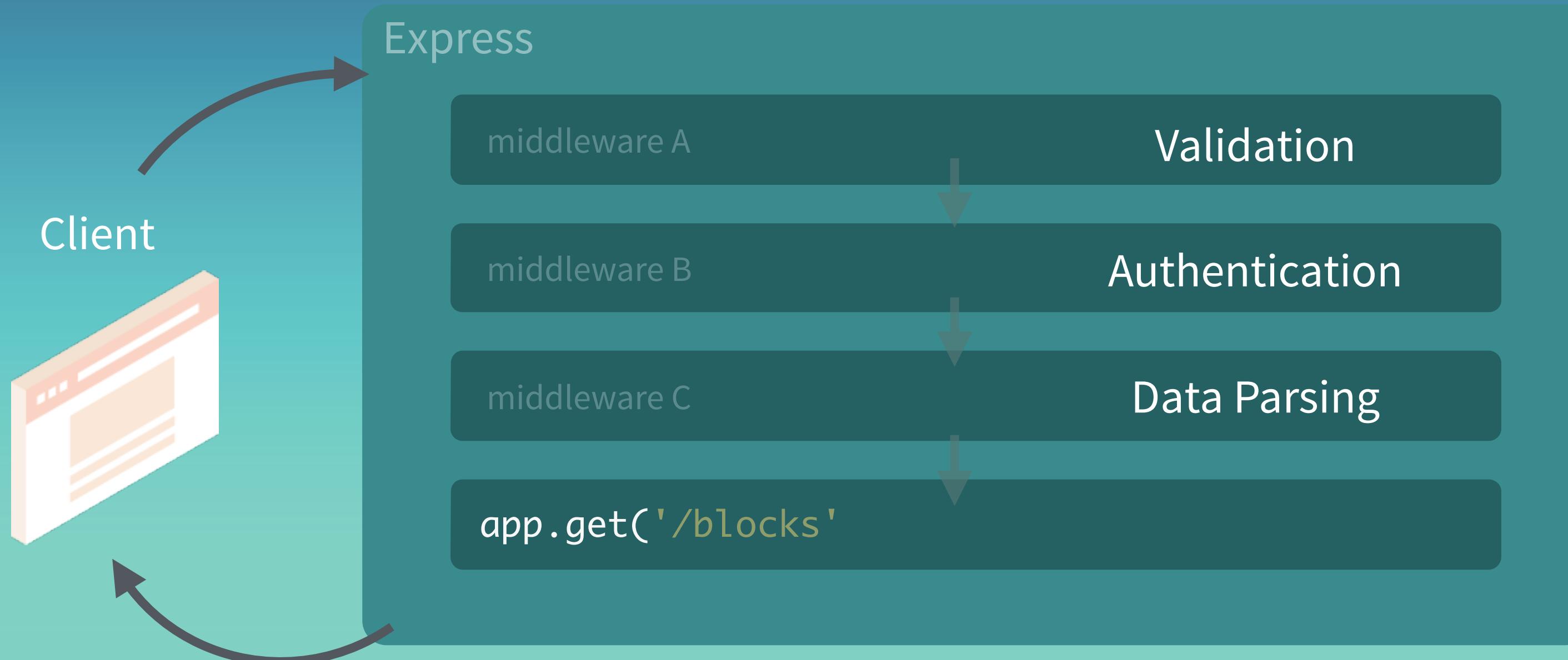
static middleware serving files
from the `public` folder

same result!



Understanding Middleware

Functions executed sequentially that access request and response



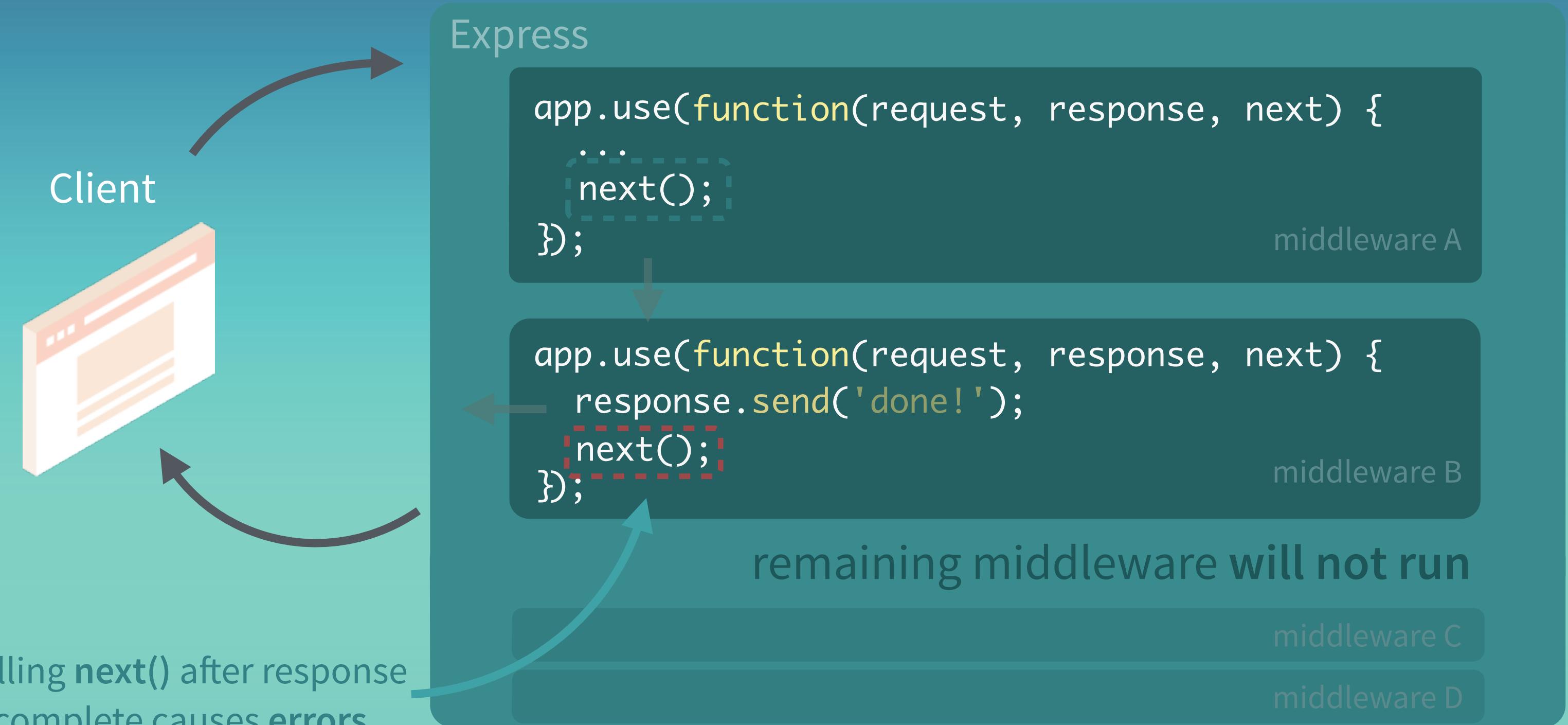
Executing Middleware functions

When `next` is called, processing moves to the next middleware.



Returning from Middleware functions

The flow stops once the response is sent back to the client



Reading the static Middleware source

The code for `static` is a good example of Express Middleware

source code from static

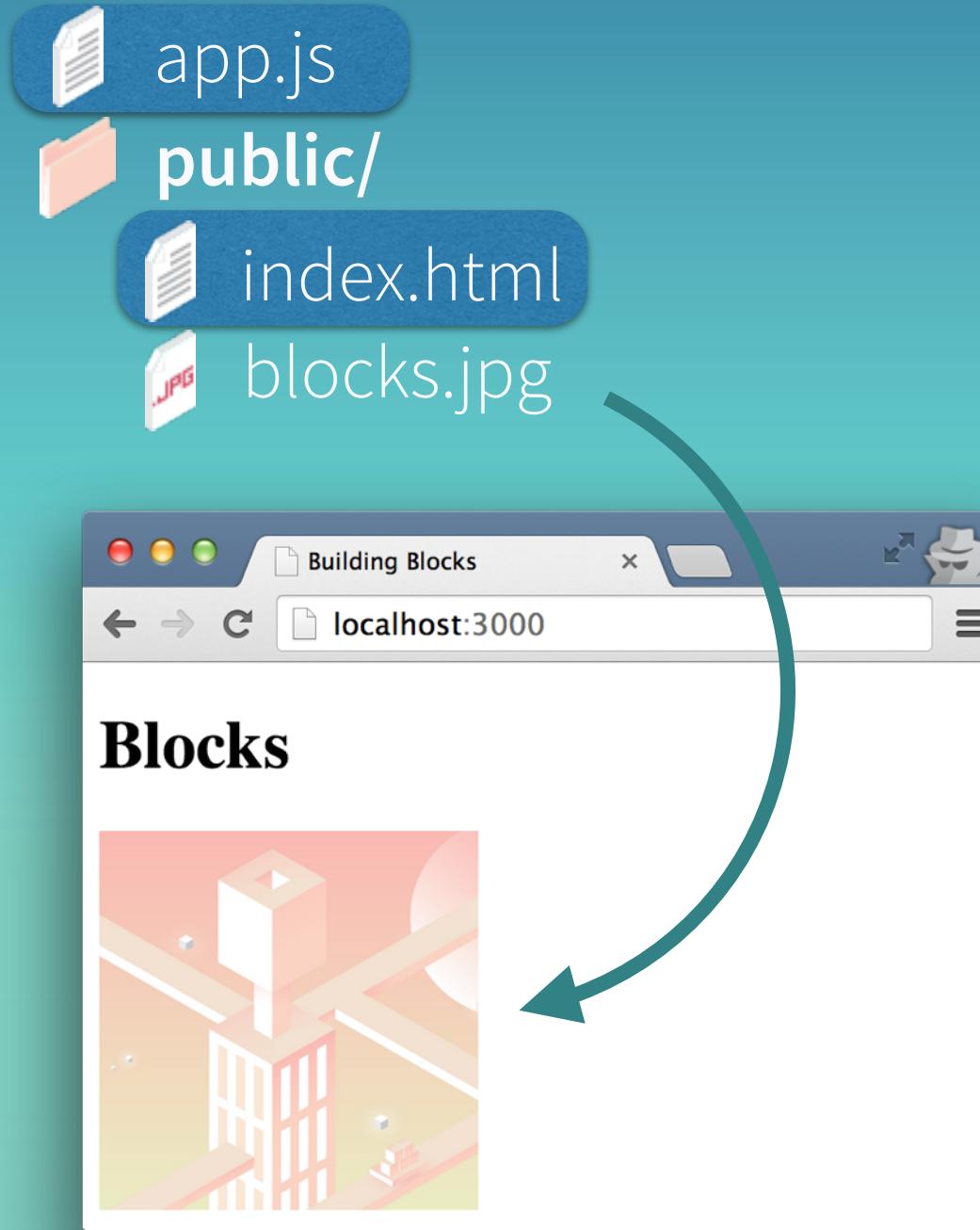
index.js

<https://github.com/expressjs/serve-static>

```
exports = module.exports = function serveStatic(root, options) {  
  ...  
  return function serveStatic(req, res, next) {  
    if (req.method !== 'GET' && req.method !== 'HEAD') {  
      return next()  
    }  
    ...  
    stream.pipe(res)  
  }  
}
```

Serving static assets

The static middleware serves **everything** under the specified folder

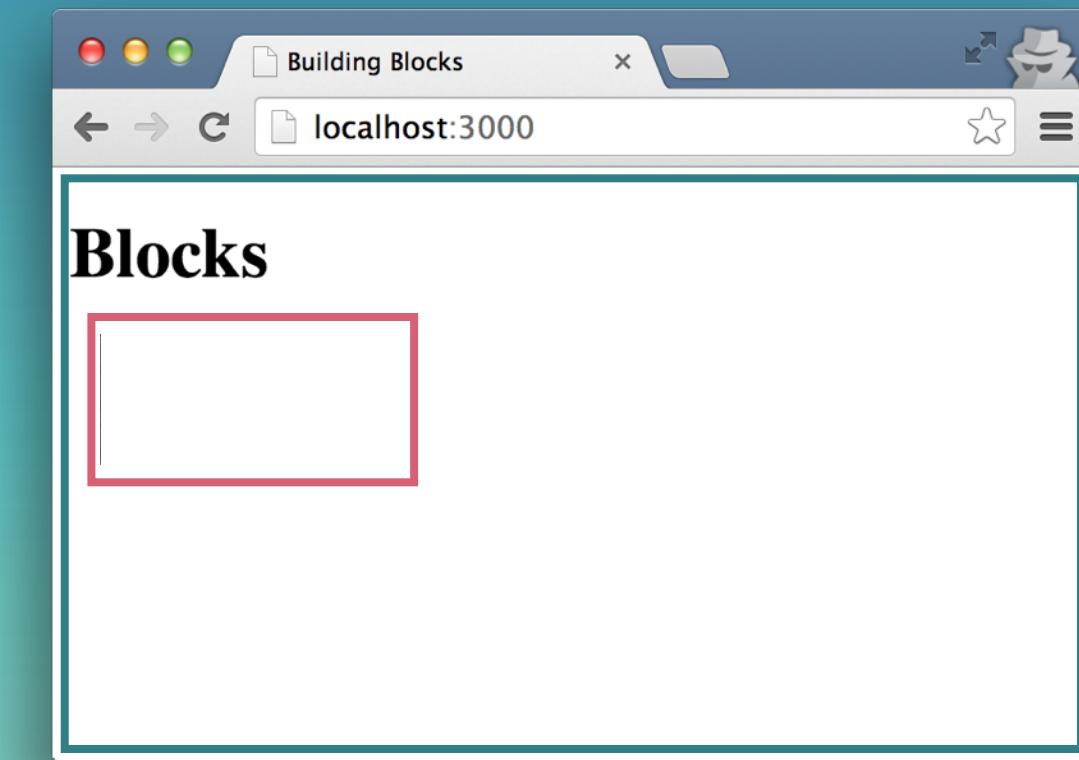
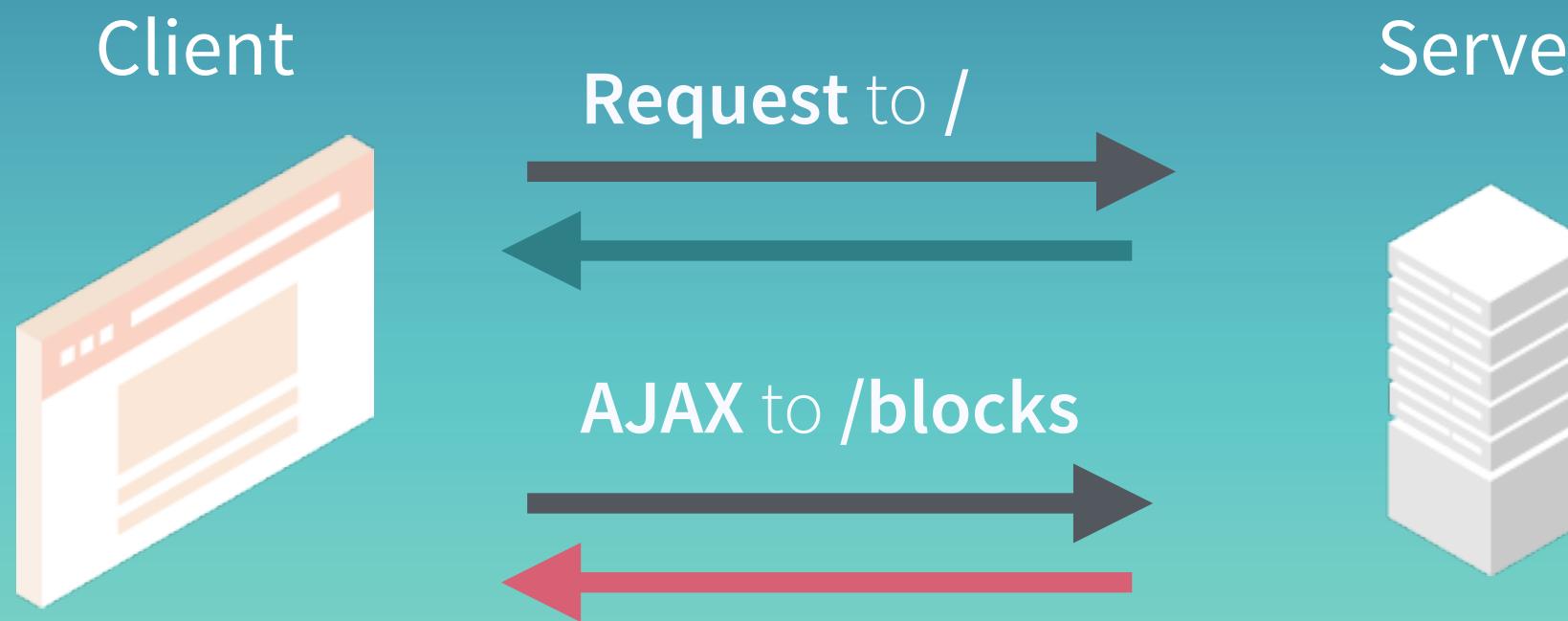


```
app.use(express.static('public'));
```

```
<!DOCTYPE html>
...
<body>
  <h1>Blocks</h1>
  <p><img src='blocks.png'></p>
</body>
</html>
```

Fetching a List of Blocks

Loading data from Express with AJAX calls



Adding client-side JavaScript

Place all files under the **public** folder

index.html



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Building Blocks</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <h1>Blocks</h1>

  <ul class='block-list'></ul>

  <script src="jquery.js"></script>
  <script src="client.js"></script>
</body>
</html>
```

Making AJAX calls

Request to **/blocks**, then append results to **block-list**



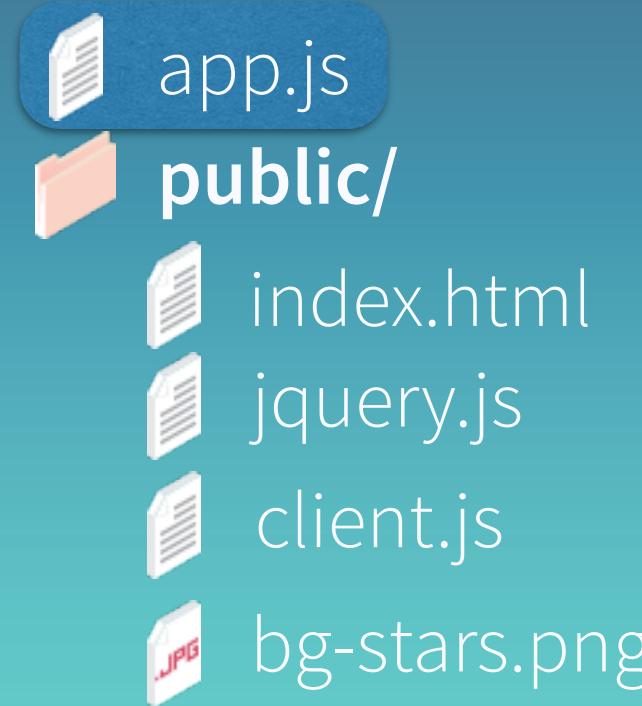
returns blocks in JSON format

client.js

```
$(function(){
    $.get('/blocks', appendToList);

    function appendToList(blocks) {
        var list = [];
        for(var i in blocks){
            list.push($('<li>', { text: blocks[i] }));
        }
        $('.block-list').append(list);
    }
});
```

Responding with JSON



app.js

```
var express = require('express');
var app = express();

app.use(express.static('public'));

app.get('/blocks', function(request, response) {
  var blocks = ['Fixed', 'Movable', 'Rotating'];
  response.json(blocks);
});

app.listen(3000);
```

Middleware

Writing Our Own

Level 2- Part II



Writing the logger module

We assign our logger function to `module.exports` in order to export it as a Node module and make it accessible from other files



```
module.exports = function(request, response, next) {  
}  
}
```

logger.js

*The Node module system follows
the CommonJS specification*

Tracking the start time for the request

We use the `Date` object to track the start time.

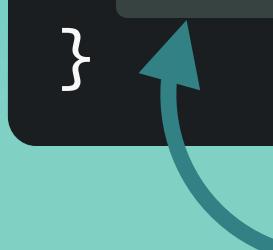


```
module.exports = function(request, response, next) {  
  var start = +new Date();
```



plus sign converts date Object
to milliseconds

```
    next();  
}
```



moves request to the **next**
middleware in the stack

logger.js

Assigning the readable stream

Standard out is a **writeable stream** which we will be writing the log to



```
logger.js  
module.exports = function(request, response, next) {  
  var start = +new Date();  
  var stream = process.stdout;  
  
  next();  
}
```

Reading the url and HTTP method

The `request` object gives us the requested URL and the HTTP method used



logger.js

```
module.exports = function(request, response, next) {  
  var start = +new Date();  
  var stream = process.stdout;  
  var url = request.url;  
  var method = request.method;  
  
  next();  
}
```

Listening for the finish event

The response object is an **EventEmitter** which we can use to **listen** to events



```
module.exports = function(request, response, next) {  
  var start = +new Date();  
  var stream = process.stdout;  
  var url = request.url;  
  var method = request.method;  
  
  response.on('finish', function() {  
    ...  
  });  
  
  next();  
}
```

logger.js

event handler function
runs **asynchronously**

the **finish** event is emitted when the
response has been handed off to the OS

Calculating the request interval



logger.js

```
module.exports = function(request, response, next) {
  var start = +new Date();
  var stream = process.stdout;
  var url = request.url;
  var method = request.method;

  response.on('finish', function() {
    var duration = +new Date() - start; ←
    ...
  });
  next();
}
```

Calculate the duration of the request

Composing the log message



logger.js

```
module.exports = function(request, response, next) {
  var start = +new Date();
  var stream = process.stdout;
  var url = request.url;
  var method = request.method;

  response.on('finish', function() {
    var duration = +new Date() - start;
    var message = method + ' to ' + url +
      '\ntook ' + duration + ' ms \n\n';
    ...
  });
  next();
}
```

Printing and moving along

We call the `write` function on the writeable stream in order to print the log



logger.js

```
module.exports = function(request, response, next) {
  var start = +new Date();
  var stream = process.stdout;
  var url = request.url;
  var method = request.method;

  response.on('finish', function() {
    var duration = +new Date() - start;
    var message = method + ' to ' + url +
      '\ntook ' + duration + ' ms \n\n';
    stream.write(message); ← prints the log message
  });

  next();
}
```

Using the logger module

We require and use our logger module in our application



app.js



logger.js



public

require and use
our module

app.js

```
var express = require('express');
var app = express();

var logger = require('./logger');
app.use(logger);

app.use(express.static('public'));

app.get('/blocks', function(request, response) {
  var blocks = ['Fixed', 'Movable', 'Rotating'];
  response.json(blocks);
});

app.listen(3000, function () {
  console.log('Listening on 3000 \n');
});
```

Reading the source for Morgan

<https://github.com/expressjs/morgan>

The screenshot shows the GitHub repository page for `expressjs/morgan`. The page title is "expressjs / morgan". The repository description is "http request logger middleware for node.js". Key statistics shown are 89 commits, 1 branch, 12 releases, and 6 contributors. The master branch is selected. A list of recent commits is displayed, including:

- 1.4.0 (dougwilson, 4 days ago)
- Fix req.ip integration when immediate: false (23 days ago)
- build: test coverage with istanbul (5 months ago)
- build: update coveralls (2 months ago)
- 1.4.0 (dougwilson, 4 days ago)
- build: move license to file (2 months ago)
- docs: fix non-https badge (4 days ago)
- Add debug messages (4 days ago)
- 1.4.0 (dougwilson, 4 days ago)

On the right side, there are links for "Code", "Issues" (0), "Pull Requests" (0), "Pulse", "Graphs", and download options ("Clone in Desktop", "Download ZIP").

User Params

Reading from the URL

Level 3 - Part I



Always returning all the Blocks

To improve efficiency, we want to be able to **limit** the number of results returned

```
var express = require('express');
var app = express();

app.get('/blocks', function(request, response) {
  var blocks = ['Fixed', 'Movable', 'Rotating'];
  response.json(blocks);
});

app.listen(3000);
```

```
$ curl http://localhost:3000/blocks
["Fixed", "Movable", "Rotating"]
```

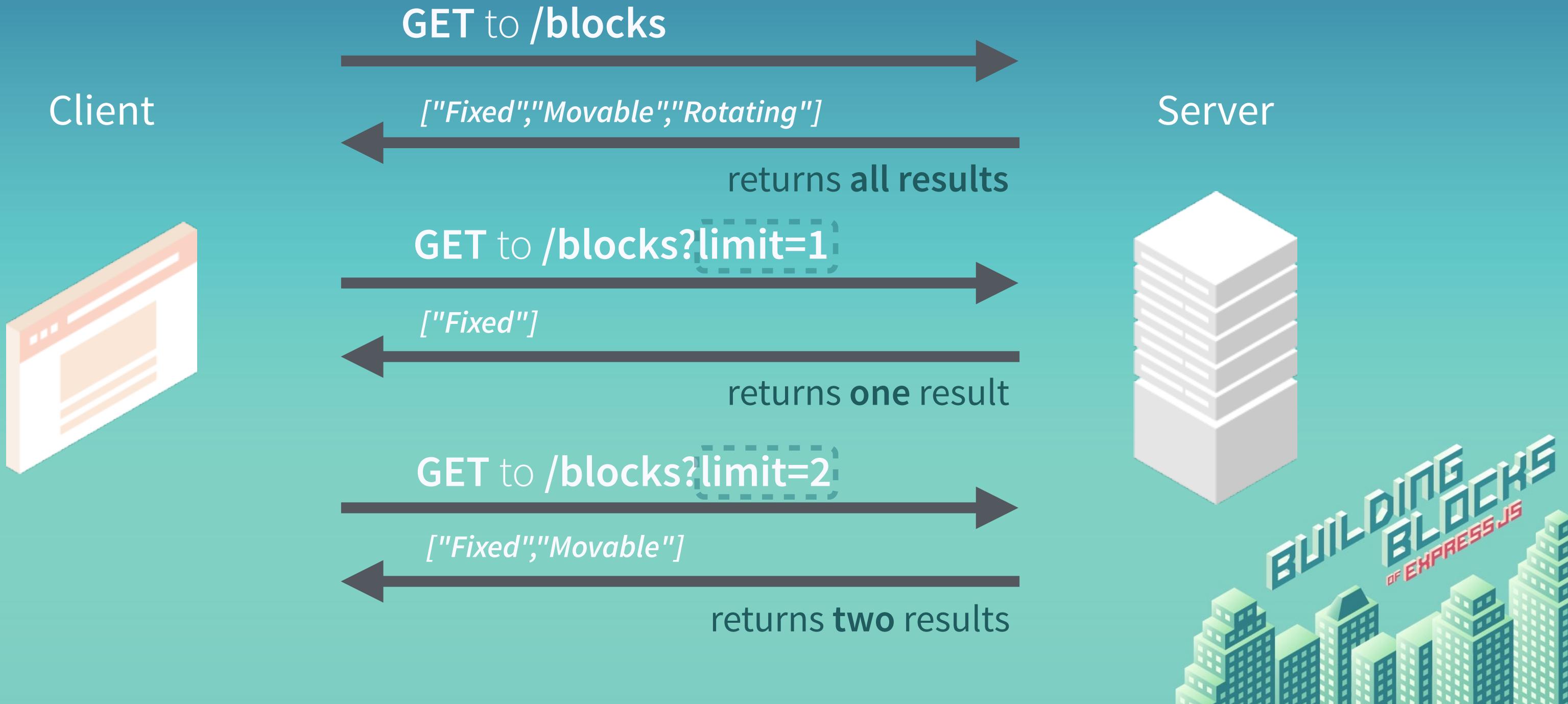
app.js

always returns
all the Blocks



Limiting the number of Blocks returned

Query strings are a great way to limit the number of results returned from an endpoint



Reading query string parameters

Use `request.query` to access query strings

app.js

```
var express = require('express');
var app = express();

app.get('/blocks', function(request, response) {
  var blocks = ['Fixed', 'Movable', 'Rotating'];
  if (request.query.limit >= 0) {
    } else {
      response.json(blocks);
    }
  });
  app.listen(3000);
```

true when a numeric value for `limit` is part of the URL

returns all results



Reading query string parameters

The slice function returns a portion of an Array

app.js

```
var express = require('express');
var app = express();

app.get('/blocks', function(request, response) {
  var blocks = ['Fixed', 'Movable', 'Rotating'];
  if (request.query.limit >= 0) {
    response.json(blocks.slice(0, request.query.limit)); ←
  } else {
    response.json(blocks);
  }
});

app.listen(3000);
```

returns limited results



Limiting results using curl

URLs with query strings can be used with curl

```
$ curl http://localhost:3000/blocks?limit=1
```

```
["Fixed"]
```

```
$ curl http://localhost:3000/blocks?limit=2
```

```
["Fixed", "Movable"]
```

```
$ curl http://localhost:3000/blocks
```

```
["Fixed", "Movable", "Rotating"]
```

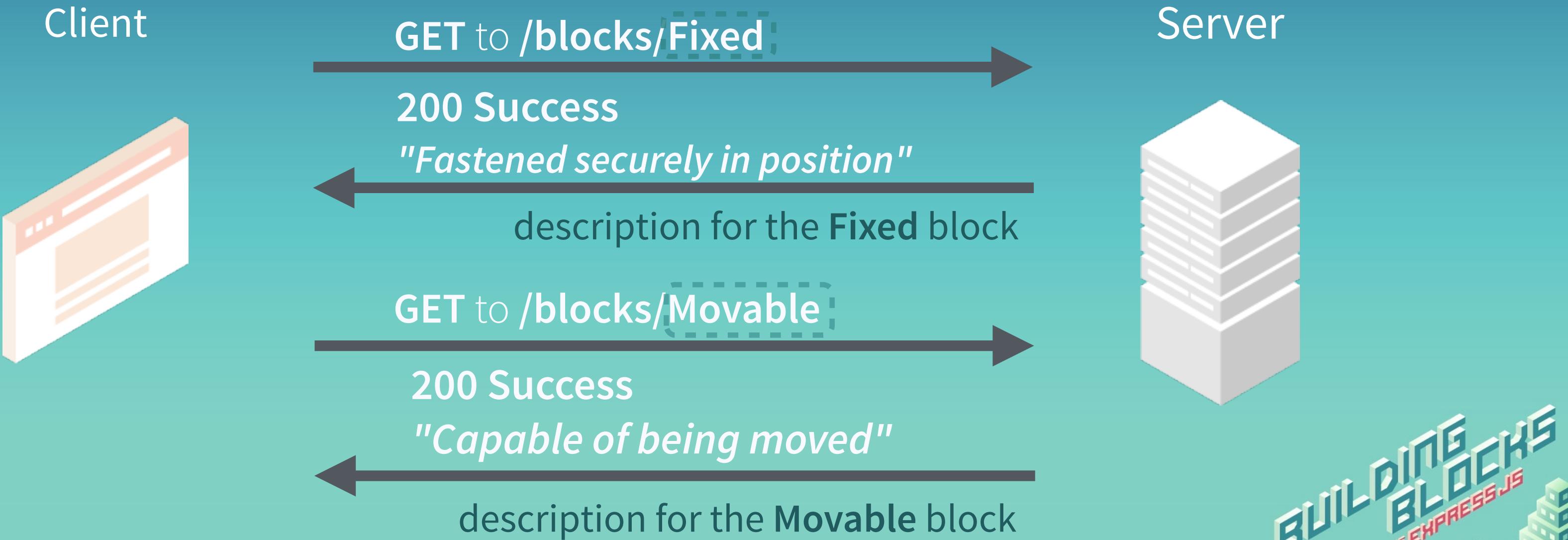
limiting results

all results when
no limit is used



Returning description for a specific Block

We can use **meaningful URLs** to return the description for specific types of Blocks



Creating Dynamic Routes

Placeholders can be used to name arguments part of the URL path

app.js

```
var express = require('express');
var app = express();

app.get('/blocks/:name', function(request, response) {
  ...
});

app.listen(3000);
```

creates **name** property on the
request.params object

request.params.name



Expanding our blocks list

In order to store additional information on blocks, we'll move from an Array to a JavaScript object

```
var express = require('express');
var app = express();

var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};

app.get('/blocks/:name', function(request, response) {
  ...
});

app.listen(3000);
```

app.js

can now be accessed
from other routes
in the file



Reading route parameters

We use `request.params.name` to look up the Block's description

app.js

```
var express = require('express');
var app = express();

var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.params.name];
  ...
});
app.listen(3000);
```



Returning block description

Responding with description and proper status code

app.js

```
var express = require('express');
var app = express();

var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.params.name];
  response.json(description);
});
app.listen(3000);
```

defaults to 200 Success
status code



Testing dynamic routes with curl

Returns proper **status codes** and response bodies

```
$ curl -i http://localhost:3000/blocks/Fixed  
HTTP/1.1 200 OK  
"Fastened securely in position"
```

The **-i** option tells curl to **include** response headers in the output

```
$ curl -i http://localhost:3000/blocks/Movable  
HTTP/1.1 200 OK  
"Capable of being moved"
```



Fixing the response for URLs not found

The status code does not indicate an **invalid URL**

```
$ curl -i http://localhost:3000/blocks/Banana
```

```
HTTP/1.1 200 OK
```

blank response

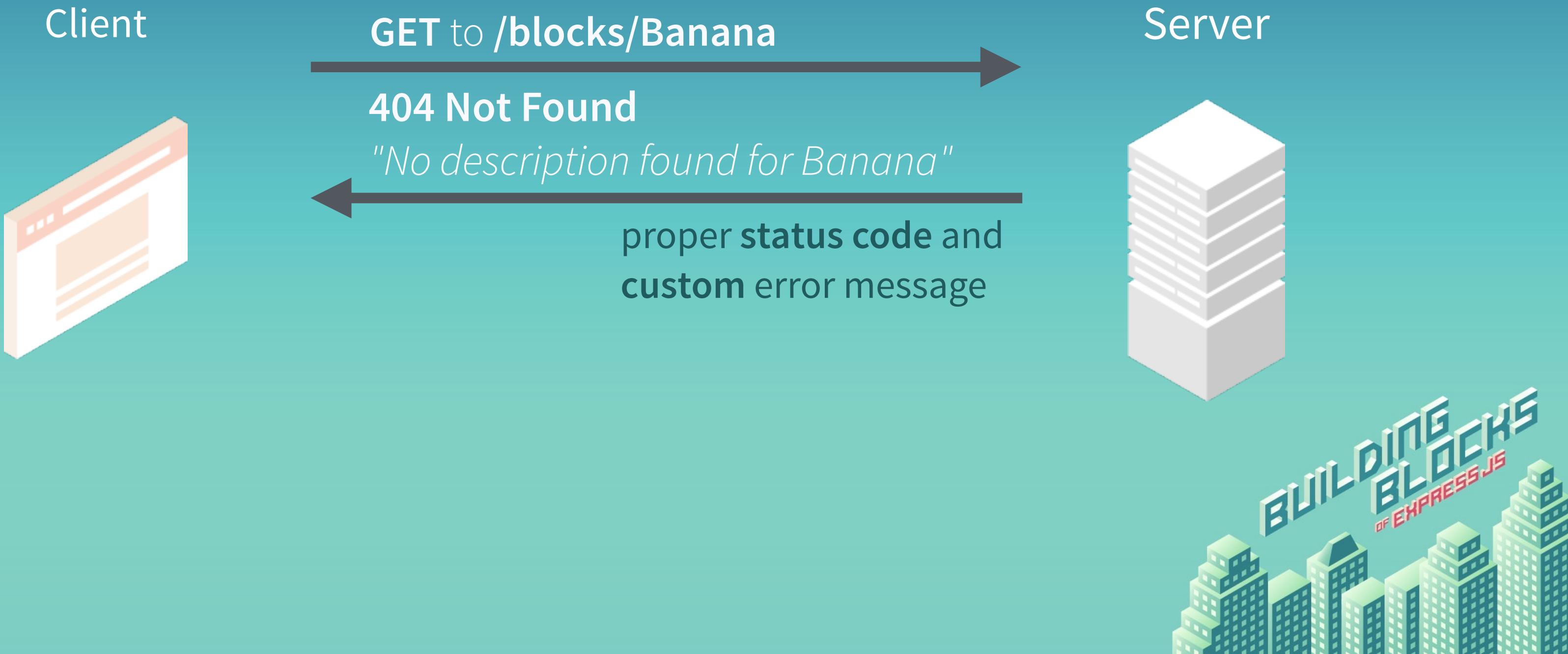
successful status code
for unsuccessful request

this Block does not exist



Handling Blocks not found

We must return a **404 Not Found** status code and an informative error message when a Block is not found



Responding from not found URLs

Trying to access non-existing properties in JavaScript objects returns **undefined**

app.js

```
...
var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.params.name];
  if (!description) {
    ...
  } else {
    response.json(description);
  }
});
```

checks for the presence of a description
to determine the response

returns **undefined** when no property
is found for a given Block name

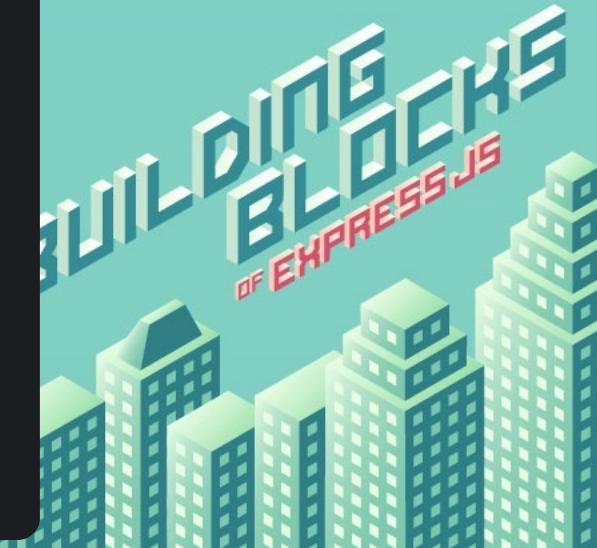


Responding with Not Found

Use the `status` function to set a custom HTTP status code

app.js

```
...
var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.params.name];
  if (!description) {
    response.status(404) ← sets the 404 Not Found
  } else {                                         status code
    response.json(description);
  }
});
```



Responding with Not Found

Respond with a custom JSON error message

app.js

```
...
var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.params.name];
  if (!description) {
    response.status(404).json('No description found for ' + request.params.name);
  } else {
    response.json(description);
  }
});
```

informative error message 

Testing invalid routes with curl

Returns proper status code and informative error message

```
$ curl -i http://localhost:3000/blocks/Banana
```

```
HTTP/1.1 404 Not Found
```

```
"No description found for Banana"
```



User Params

Massaging user data

Level 3 - Part II



Routes don't match all cases

Current implementation only matches on **exact** Block name

```
$ curl -i http://localhost:3000/blocks/Fixed
```

```
HTTP/1.1 200 0K  
"Fastened securely in position"
```

```
$ curl -i http://localhost:3000/blocks/fixed
```

```
HTTP/1.1 404 Not Found  
"No description found for fixed"
```



does not match
on lower case



Normalizing the request parameter

Let's split the steps to improve code clarity

app.js

```
...
var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.params.name];
  ...
});
...
}
```

doing two things at once



Normalizing the request parameter

When one line does only **one thing**, it makes code easier to understand

app.js

```
...
var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};
app.get('/blocks/:name', function(request, response) {
  var name = request.params.name;
  var block = name[0].toUpperCase() + name.slice(1).toLowerCase();
  ...
});
```

first character to upper case
and remaining characters to
lowercase



Normalizing the request parameter

Use the **normalized** block name to look up its description

```
app.js  
...  
  
var blocks = {  
  'Fixed': 'Fastened securely in position',  
  'Movable': 'Capable of being moved',  
  'Rotating': 'Moving in a circle around its center'  
};  
  
app.get('/blocks/:name', function(request, response) {  
  var name = request.params.name;  
  var block = name[0].toUpperCase() + name.slice(1).toLowerCase();  
  var description = blocks[block];  
  if (!description) {  
    ...  
  }  
});  
...
```

block **name** is now in the same
format as the properties in the
blocks object



app.js



BUILDING BLOCKS
OF EXPRESS.JS

Supporting any url argument case

```
$ curl -i http://localhost:3000/blocks/Fixed
```

HTTP/1.1 200 OK

"Fastened securely in position"

```
$ curl -i http://localhost:3000/blocks/fixed
```

HTTP/1.1 200 OK

"Fastened securely in position"

```
$ curl -i http://localhost:3000/blocks/fiXeD
```

HTTP/1.1 200 OK

"Fastened securely in position"

any case is now properly supported



Same parameter used on multiple routes

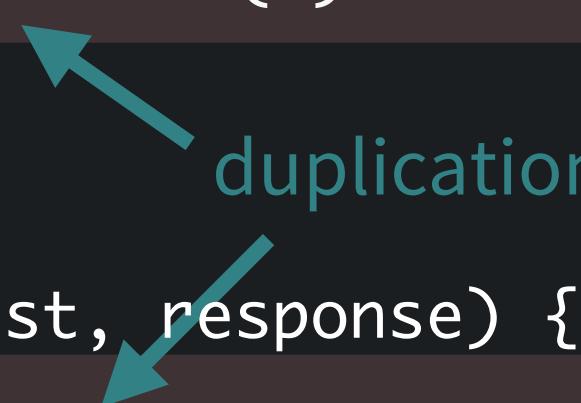
app.js

```
var blocks = { ... };

var locations = {
  'Fixed': 'First floor', 'Movable': 'Second floor', 'Rotating': 'Penthouse'
};

app.get('/blocks/:name', function(request, response) {
  var name = request.params.name;
  var block = name[0].toUpperCase() + name.slice(1).toLowerCase();
  ...
});

app.get('/locations/:name', function(request, response) {
  var name = request.params.name;
  var block = name[0].toUpperCase() + name.slice(1).toLowerCase();
  ...
});
```



duplication

Extracting duplication to app.param

The `app.param` function maps placeholders to callback functions.
It's useful for running **pre-conditions** on dynamic routes.

app.js

```
var blocks = { ... };

var locations = {
  'Fixed': 'First floor', 'Movable': 'Second floor', 'Rotating': 'Penthouse'
};

app.param('name', function(request, response, next) {
  var name = request.params.name;
  var block = name[0].toUpperCase() + name.slice(1).toLowerCase();
  ...
});
```

called for routes which include the `:name` placeholder



Setting properties on the request object

Properties set on the `request` object can be accessed from all subsequent routes in the application

app.js

```
var blocks = { ... };

var locations = {
  'Fixed': 'First floor', 'Movable': 'Second floor', 'Rotating': 'Penthouse'
};

app.param('name', function(request, response, next) {
  var name = request.params.name;
  var block = name[0].toUpperCase() + name.slice(1).toLowerCase();

  request.blockName = block; ← can be accessed from other routes
                           in the application

  next(); ← must be called to resume request
});
```

Accessing custom properties on request

We can read properties on **request** which were set on **app.param**

```
app.js

...
app.param('name', function(request, response, next) {
  ...
});

app.get('/blocks/:name', function(request, response) {
  var description = blocks[request.blockName];
  ...
});

app.get('/locations/:name', function(request, response) {
  var location = locations[request.blockName];
  ...
});
```

Dynamic routes with curl

Refactoring improved our code without affecting the output

```
$ curl -i http://localhost:3000/blocks/fixed
```

```
HTTP/1.1 200 OK
```

```
"Fastened securely in position"
```

```
$ curl -i http://localhost:3000/locations/fixED
```

```
HTTP/1.1 200 OK
```

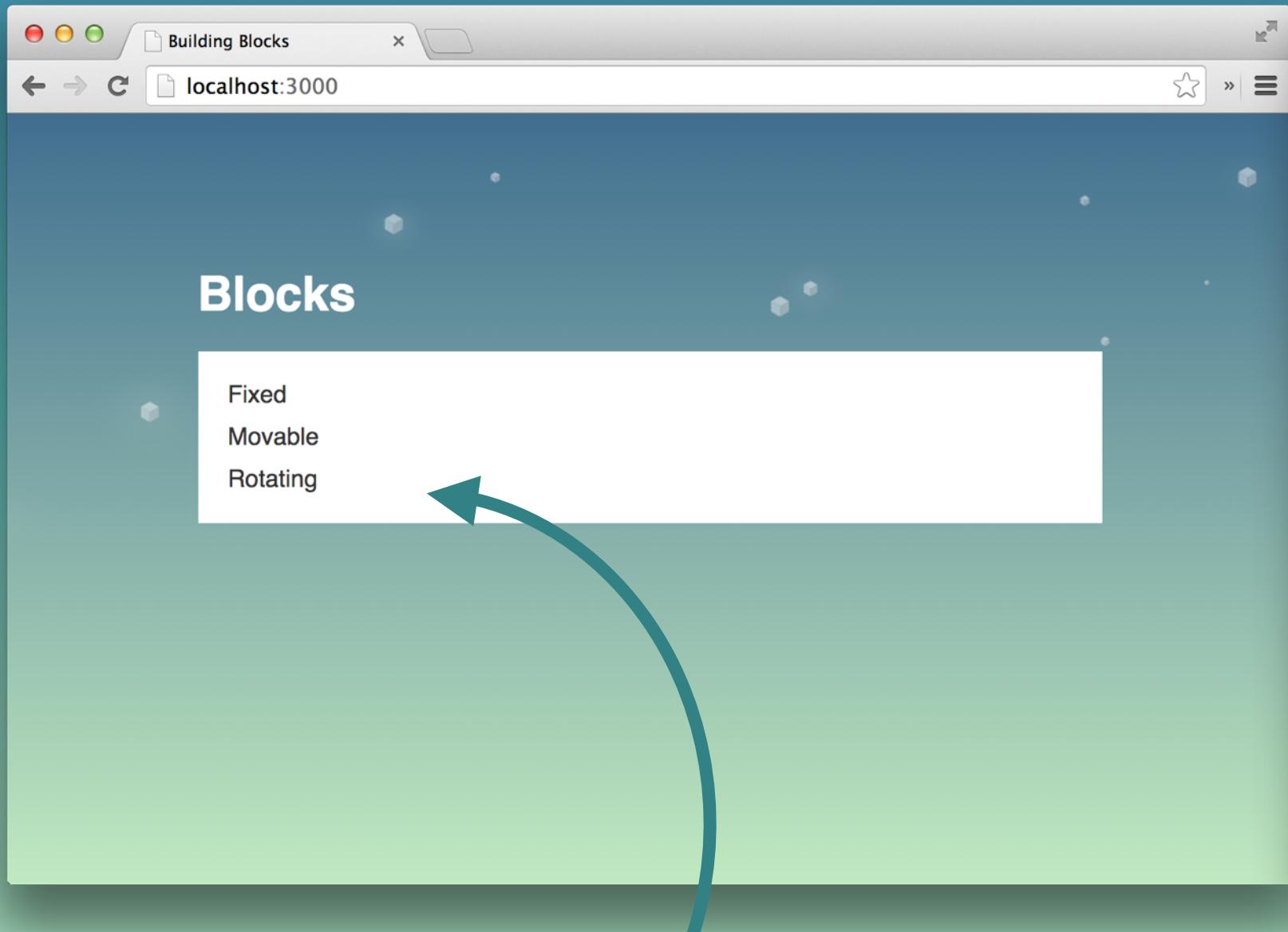
```
"First floor"
```

same result
as before



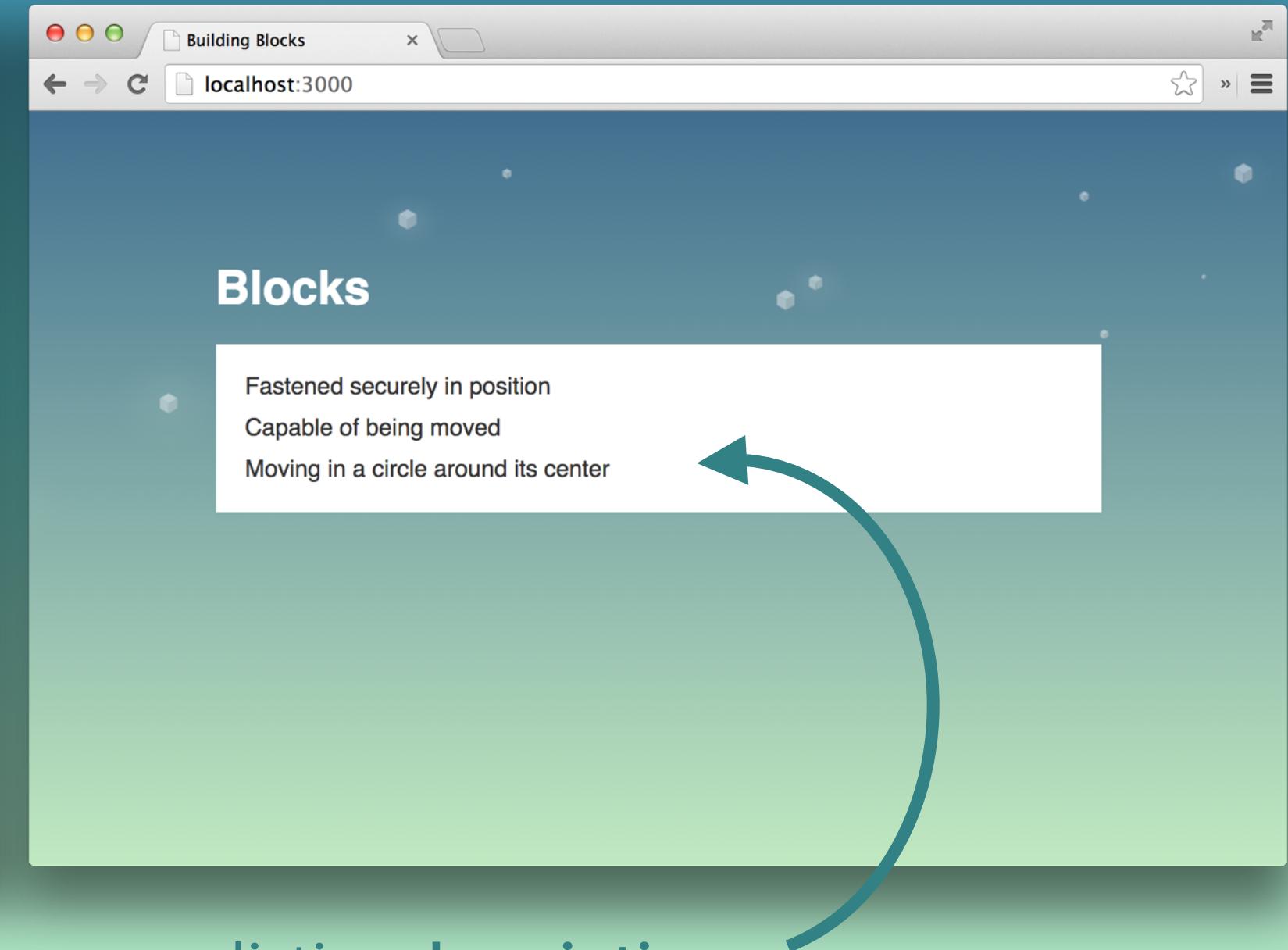
Something looks different

Initially:



listing block names

Now:



listing descriptions
instead of names

Breaking the initial listing of Blocks names

Initially:

```
var blocks = ['Fixed', 'Movable', 'Rotating'];
```

app.js

Now:

```
var blocks = {  
  'Fixed': 'Fastened securely in position',  
  'Movable': 'Capable of being moved',  
  'Rotating': 'Moving in a circle around its center'  
};
```

moved from Array to object

app.js



Fixing Block names

Responding with **object** instead of **Array** is what broke our route

app.js

```
var blocks = {  
  'Fixed': 'Fastened securely in position',  
  'Movable': 'Capable of being moved',  
  'Rotating': 'Moving in a circle around its center'  
};  
  
app.get('/blocks', function(request, response) {  
  response.json(blocks);  
});  
  
...
```

serializes blocks object



Fixing Block names

The `Object.keys` function returns an Array with the object's properties

```
app.js
```

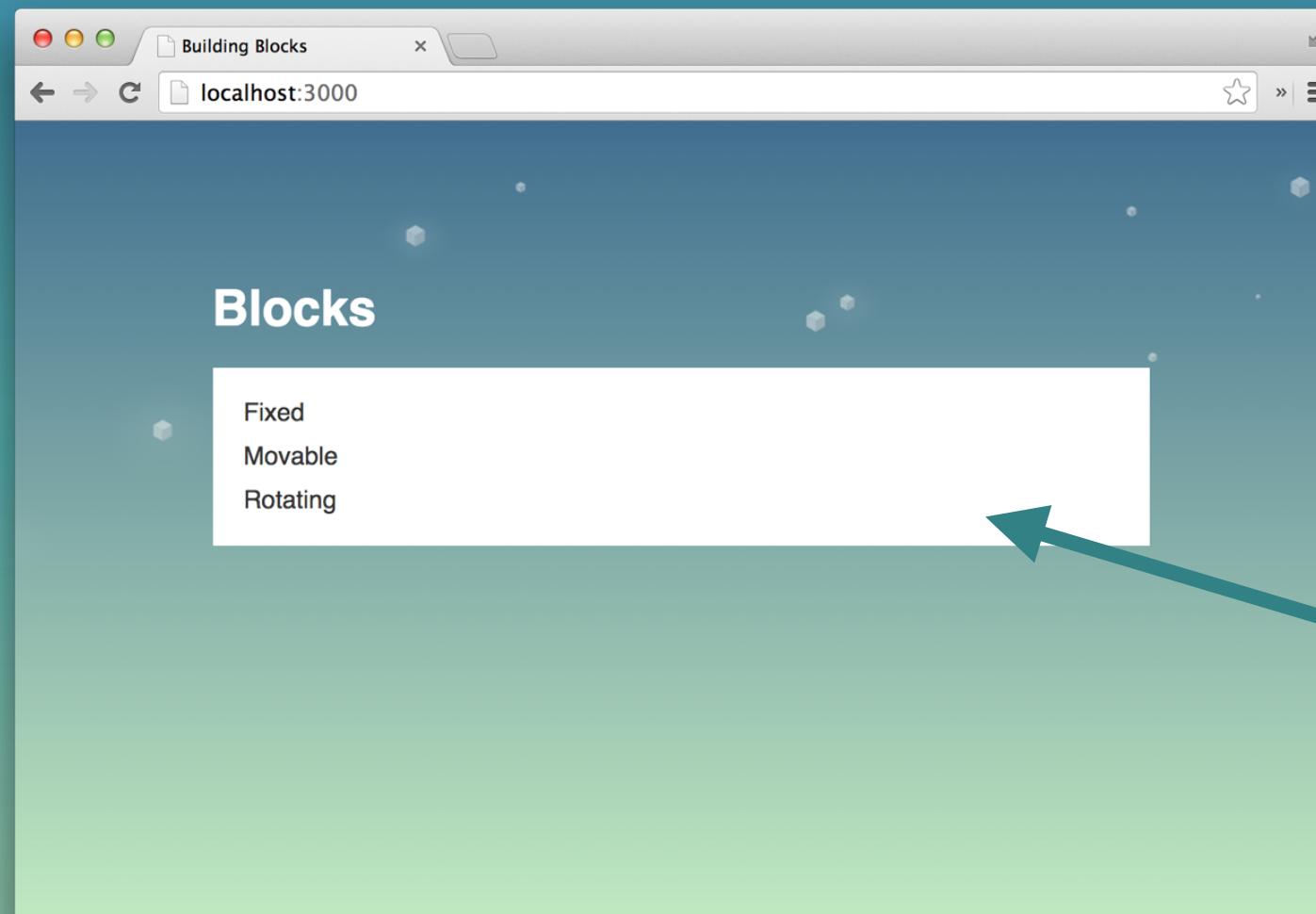
```
var blocks = {  
  'Fixed': 'Fastened securely in position',  
  'Movable': 'Capable of being moved',  
  'Rotating': 'Moving in a circle around its center'  
};  
  
app.get('/blocks', function(request, response) {  
  response.json(Object.keys(blocks));  
});  
  
...
```

returns properties from
the blocks object



Responding with Block names

Now:



```
$ curl -i http://localhost:3000(blocks
```



```
HTTP/1.1 200 OK
```

```
["Fixed", "Movable", "Rotating"]
```

back to listing
Block names

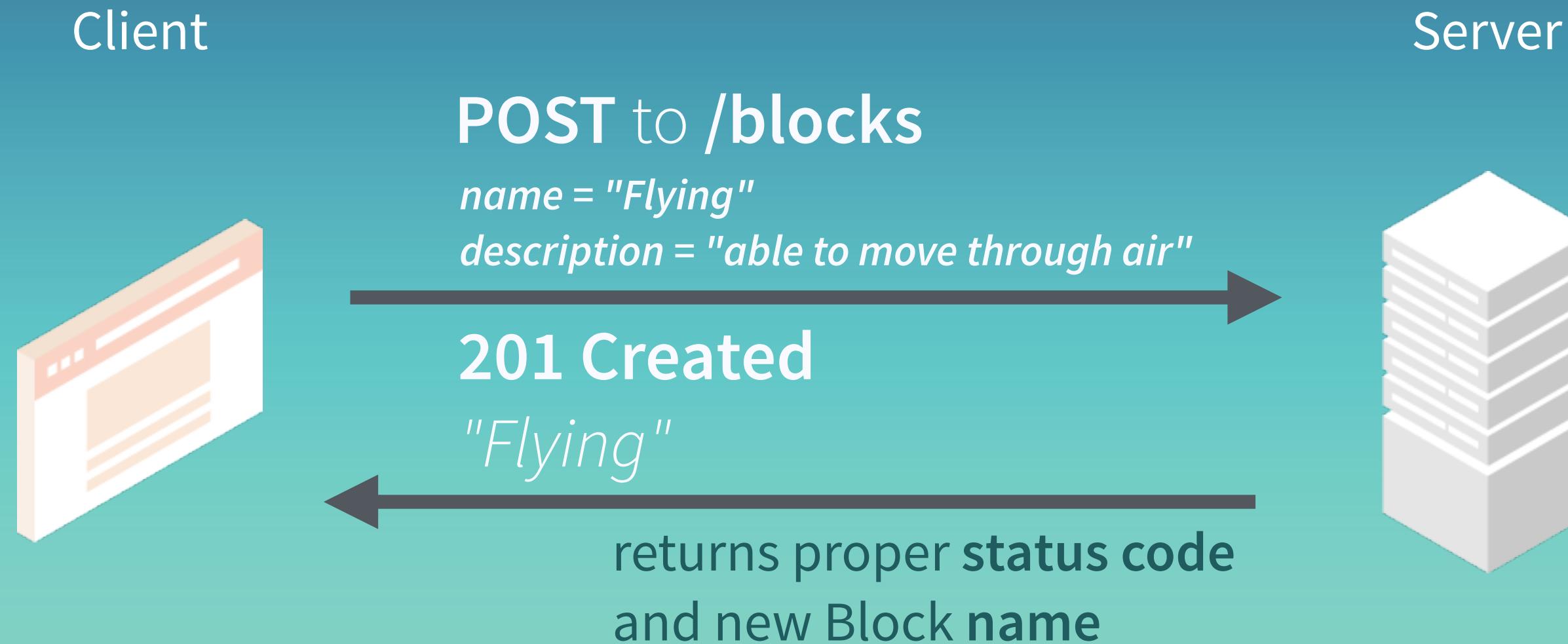


POST Requests

Level 4 - Part I



Creating new Blocks



Adding a form to index.html

Text field inputs will be needed for name and description



we'll define form
attributes in JavaScript

```
index.html
```

```
...  
<body>  
  
<h1>Blocks</h1>  
  
<form>  
  <legend>New Block</legend>  
  <input name="name" placeholder="Name"><br/>  
  <input name="description" placeholder="Description">  
  <input type="Submit">  
</form>  
  
<ul class='block-list'></ul>  
...
```

A large black callout box contains the code for the 'index.html' page. It highlights the 'form' tag and its contents with a light gray rounded rectangle. A teal arrow points from the word 'form' in the explanatory text on the left towards the highlighted area in the code.

Submitting the form with JavaScript

Data is sent in a POST request to the `/blocks` endpoint

- app.js
- public/
 - index.html
 - jquery.js
 - client.js
 - style.css
 - bg-stars.png

client.js

```
$(function(){
  $.get('/blocks', appendToList);

  ...
  $('form').on('submit', function(event) {
    event.preventDefault();
    var form = $(this);
    var blockData = form.serialize(); // recently created block // transforms form data to URL-encoded notation

    $.ajax({
      type: 'POST', url: '/blocks', data: blockData
    }).done(function(blockName){
      ...
    });
  });
});
```

Updating the list with the new Block

We'll reuse the `appendToList` function from earlier to add new blocks to the list

- app.js
- public/
- index.html
- jquery.js
- client.js**
- style.css
- bg-stars.png

```
$function(){
  $.get('/blocks', appendToList);
  ...
  $('form').on('submit', function(event) {
    event.preventDefault();
    var form = $(this);
    var blockData = form.serialize();

    $.ajax({
      type: 'POST', url: '/blocks', data: blockData
    }).done(function(blockName){
      appendToList(
        );
    );
  );
  ...
});
```

client.js

same function being called

Updating the list with the new Block

The `appendToList` function expects an array of Blocks

client.js

-  app.js
-  public/
 -  index.html
 -  jquery.js
 -  client.js
-  style.css
-  bg-stars.png

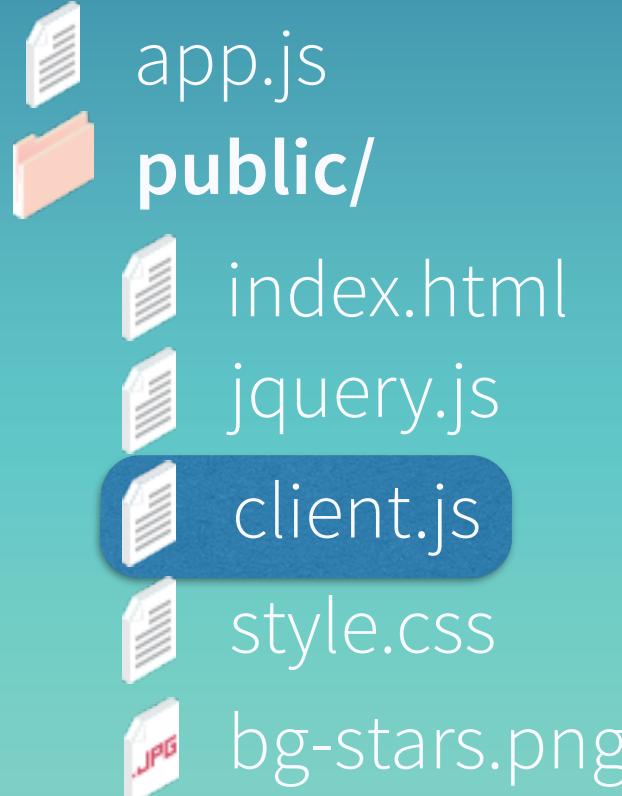
```
$(function(){
  $.get('/blocks', appendToList);
  ...
  $('form').on('submit', function(event) {
    event.preventDefault();
    var form = $(this);
    var blockData = form.serialize();

    $.ajax({
      type: 'POST', url: '/blocks', data: blockData
    }).done(function(blockName){
      appendToList([blockName]); ←
    });
  });
  ...
});
```

array with the new block
as its single argument

Clearing input fields after submission

We must clear the input text fields after posting the form



client.js

```
$(function(){
  ...
  $('form').on('submit', function(event) {
    event.preventDefault();
    var form = $(this);
    var blockData = form.serialize();

    $.ajax({
      type: 'POST', url: '/blocks', data: blockData
    }).done(function(blockName){
      appendToList([blockName]);
      form.trigger('reset'); // Cleans up form
    });
  });
  ...
});
```

client.js

The code shows a jQuery plugin for a form. It prevents the default submission behavior, serializes the form data, sends a POST request to '/blocks' with the data, and then triggers a 'reset' event on the form. A callout points to the 'reset' trigger line with the text "cleans up form text input fields".

Adding links to Blocks

client.js



link to each Block's
description

```
$(function(){
  ...
  $('form').on('submit', function(event) {
    ...
  });

  function appendToList(blocks) {
    var list = [];
    var content, block;
    for(var i in blocks){
      block = blocks[i];
      content = '<a href="/blocks/' + block + '">' + block + '</a>';
      list.push($('<li>', { html: content }));
    }
    $('.block-list').append(list)
  });
});
```

A teal arrow points from the text "link to each Block's description" in the sidebar to the line of code "content = '' + block + '';' in the client.js code block.

Posting

Parsing depends on a middleware **not** shipped with Express



\$ npm install body-parser

```
var express = require('express');
var app = express();

var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = { ... };

...
```

app.js

forces the use of the native
querystring Node library

Creating a POST route

Routes can take multiple handlers as arguments and will call them sequentially

app.js

```
var express = require('express');
var app = express();

var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = { ... };

app.post('/blocks', parseUrlencoded, function(request, response) {
});
```



Using multiple route handlers is useful for re-using middleware that load resources, perform validations, authentication, etc.

Reading request data

Form submitted data can be accessed through `request.body`

app.js

```
var express = require('express');
var app = express();

var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = { ... };

app.post('/blocks', parseUrlencoded, function(request, response) {
  var newBlock = request.body;
  ...
});
```



returns form data

Creating a new Block

The form elements are parsed to object properties, **name** and **description**

app.js

```
var express = require('express');
var app = express();

var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = { ... };

app.post('/blocks', parseUrlencoded, function(request, response) {
  var newBlock = request.body;
  blocks[newBlock.name] = newBlock.description;
});

...
```

adds new block
to the blocks object

Responding from a POST request

The response includes proper **status code** and the block **name**

app.js

```
var express = require('express');
var app = express();

var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = { ... };

app.post('/blocks', parseUrlencoded, function(request, response) {
  var newBlock = request.body;
  blocks[newBlock.name] = newBlock.description;

  response.status(201).json(newBlock.name);
});
```

...

 sets the 201 Created status code

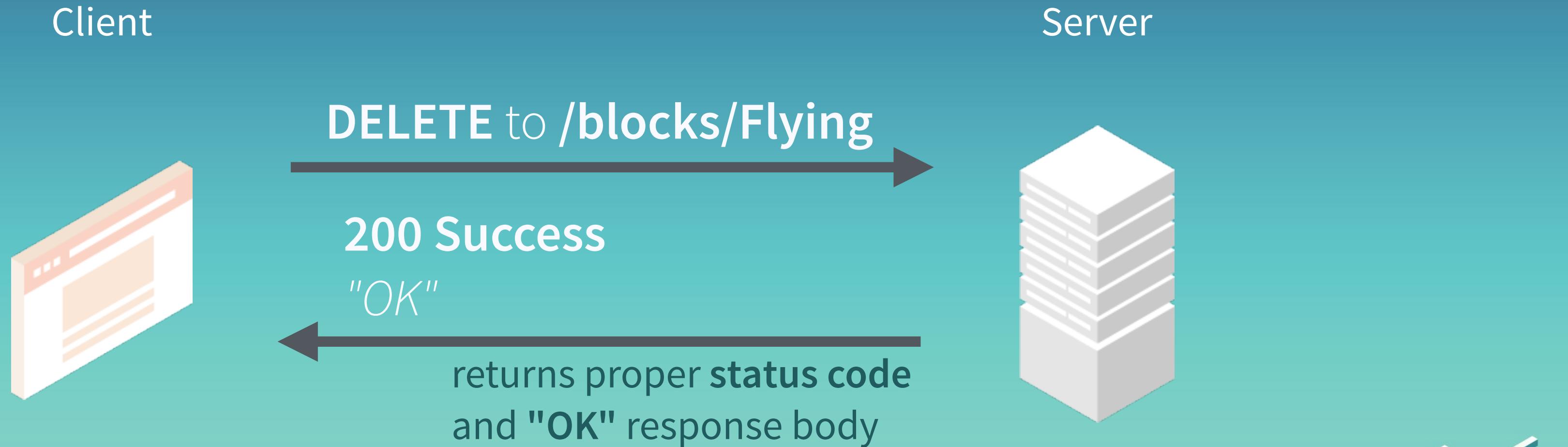
 responds with
new block **name**

DELETE Requests

Level 4 - Part II

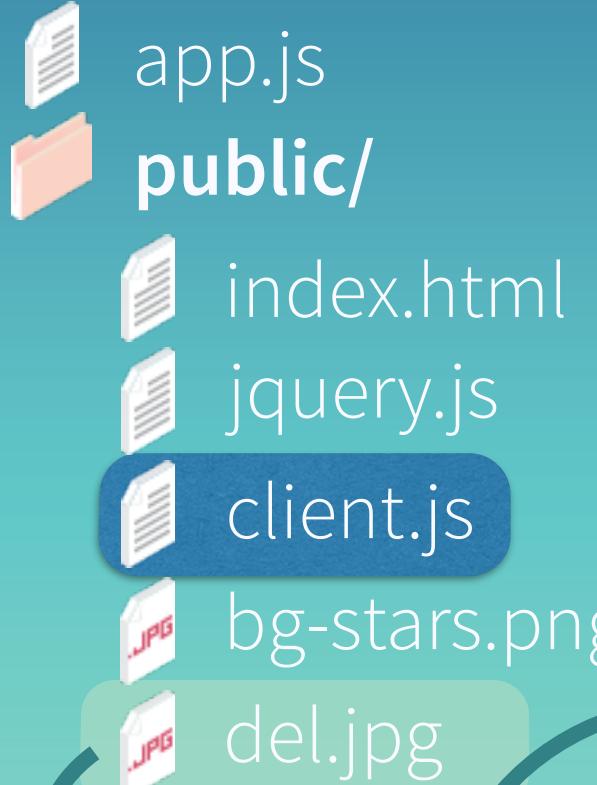


Deleting existing Blocks



Adding delete links to Blocks list

client.js



```
$(function(){
  ...
  function appendToList(blocks) {
    var list = [];
    var content, block;
    for(var i in blocks){
      block = blocks[i];
      content = '<a href="/blocks/'+block+'">' + block + '</a> ' +
      '<a href="#" data-block="'+block+'></a>';
      list.push($('- ', { html: content }));
    }
    $('.block-list').append(list)
  });
});

```

Listening for click events

Let's attach an event listener on all links with a **data-block** attribute



```
$(function(){
  ...
  $('.block-list').on('click', 'a[data-block]', function(event){
    ...
  });
});
```

links with a **data-block** attribute

- ✖ Fixed
- ✖ Movable
- ✖ Rotating

client.js

Making DELETE request to /blocks

client.js

- app.js
- public/
 - index.html
 - jquery.js
 - client.js
 - bg-stars.png
 - del.jpg

```
$(function() {  
  ...  
  $('.block-list').on('click', 'a[data-block]', function(event){  
    if (!confirm('Are you sure ?')) {  
      return false;  
    }  
  
    var target = $(event.currentTarget);  
  
    $.ajax({  
      type: 'DELETE', url: '/blocks/' + target.data('block')  
    }).done(function() {  
      ...  
    });  
  });  
});
```

the link element
that was clicked

reads the block name from
the link's **data-block** attribute

Removing elements from the page

client.js

-  app.js
-  public/
 -  index.html
 -  jquery.js
 -  client.js
-  bg-stars.png
-  del.jpg

```
$(function(){
  ...
  $('.block-list').on('click', 'a[data-block]', function(event){
    if (!confirm('Are you sure ?')) {
      return false;
    }

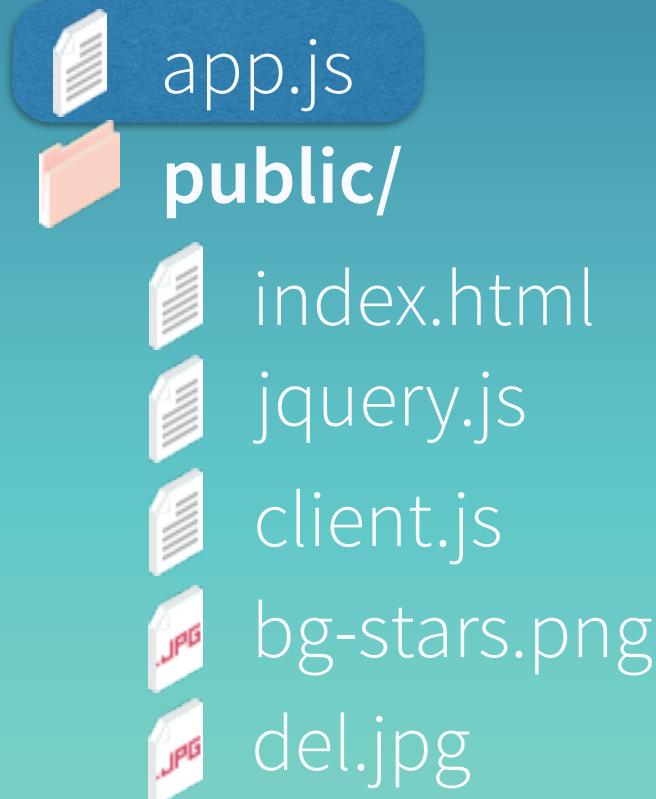
    var target = $(event.currentTarget);

    $.ajax({
      type: 'DELETE', url: '/blocks/' + target.data('block')
    }).done(function() {
      target.parents('li').remove();
    });
  });
});
```

removes li element containing the link

Creating a DELETE route

The delete route takes the block name as argument



```
app.js
```

```
var express = require('express');
var app = express();

var blocks = { ... };

app.delete('/blocks/:name', function(request, response) {
  ...
});
```

Deleting Blocks

The `delete` operator from JavaScript removes a property from an object

app.js

```
var express = require('express');
var app = express();

var blocks = { ... };

app.delete('/blocks/:name', function(request, response) {
  delete blocks[request.blockName];
});

...
```

removes entry from
the blocks object

in case you don't remember,
this is where we set `blockName`

```
app.param('name', ...)
```

Responding with sendStatus

The `sendStatus` function sets both the status code and the response body

app.js

```
var express = require('express');
var app = express();

var blocks = { ... };

app.delete('/blocks/:name', function(request, response) {
  delete blocks[request.blockName];
  response.sendStatus(200);
});
```



also sets response
body to “OK”

Route Instances

Level 5 - Part I



Repetition in route names

All routes seem to be handling requests to similar paths...

app.js

```
var express = require('express');
var app = express();
...
app.get('/blocks', function(request, response) { ← identical path
  ...
});
app.get('/blocks/:name', function(request, response) { ← identical path
  ...
});
app.post('/blocks', parseUrlencoded, function(request, response) { ← identical path
  ...
});
app.delete('/blocks/:name', function(request, response) { ← identical path
  ...
});
...
});
```

Replacing repetition with a route instance

Using `app.route` is a recommended approach for avoiding duplicate route names

app.js

```
var express = require('express');
var app = express();
...
var blocksRoute = app.route('/blocks')
...
app.listen(3000);
```

returns route object which
handles all requests to
the /blocks path

Routes that act on /blocks

No path argument is needed for route handlers belonging to the same route instance

```
app.js
var express = require('express');
var app = express();
...
var blocksRoute = app.route('/blocks')
blocksRoute.get(function(request, response) {
  ...
});
blocksRoute.post(parseUrlencoded, function(request, response) {
  ...
});
...
app.listen(3000);
```

The diagram illustrates the flow of routes from the main application file (`app.js`) to a specific route instance (`blocksRoute`). A teal arrow labeled "used to be" points from the `app.get('/blocks' ...)` call in `app.js` to the `blocksRoute.get` method. Another teal arrow labeled "used to be" points from the `app.post('/blocks' ...)` call in `app.js` to the `blocksRoute.post` method.

Removing intermediate variables

There's unnecessary repetition of the `blocksRoute` variable

app.js

```
var express = require('express');
var app = express();
...
var blocksRoute = app.route('/blocks')
blocksRoute.get(function(request, response) {
    ...
});
blocksRoute.post(parseUrlencoded, function(request, response) {
    ...
});
...
app.listen(3000);
```

Chaining function calls on route

Chaining functions can eliminate **intermediate variables** and help our code stay more **readable**. This is a pattern commonly found in Express applications.

app.js

```
var express = require('express');
var app = express();
...
app.route('/blocks')
  .get(function(request, response) {
    ...
  });
  .post(parseUrlencoded, function(request, response) {
    ...
  });
  ...
}
```

no semi-colon at
the end of the line

lines starting with **dot** indicate function calls
on the object returned from the **previous line**

chaining means calling
functions on the return value of
previous functions

Dynamic route instances

The `app.route` function accepts the same url argument format as before

app.js

```
var express = require('express');
var app = express();
...
app.route('/blocks')
  .get(function(request, response) {
    ...
  });
  .post(parseUrlencoded, function(request, response) {
    ...
  });
app.route('/blocks/:name')
```

returns route object which handles all
requests to the `/blocks/:name` path

Routes that act on /blocks/:name

Our route handlers for /blocks/:name reference **blocks** fetched by their name

app.js

```
var express = require('express');
var app = express();
...
app.route('/blocks')
...
app.route('/blocks/:name')
  .get(function(request, response) {
    ...
  })
  .delete(function(request, response) {
    ...
  });
app.listen(3000);
```

used to be

app.get('/blocks/:name/' ...)

app.delete('/blocks/:name'...)

used to be

Route Files

Level 5 - Part II



Single application file is too long

app.js

```
var express = require('express');
...
app.route('/blocks')
  .get(function(request, response) {
    ...
  });
  .post(parseUrlencoded, function(request, response) {
    ...
  });
app.route('/blocks/:name')
  .get(function(request, response) {
    ...
  })
  .delete(function(request, response) {
    ...
  });
...

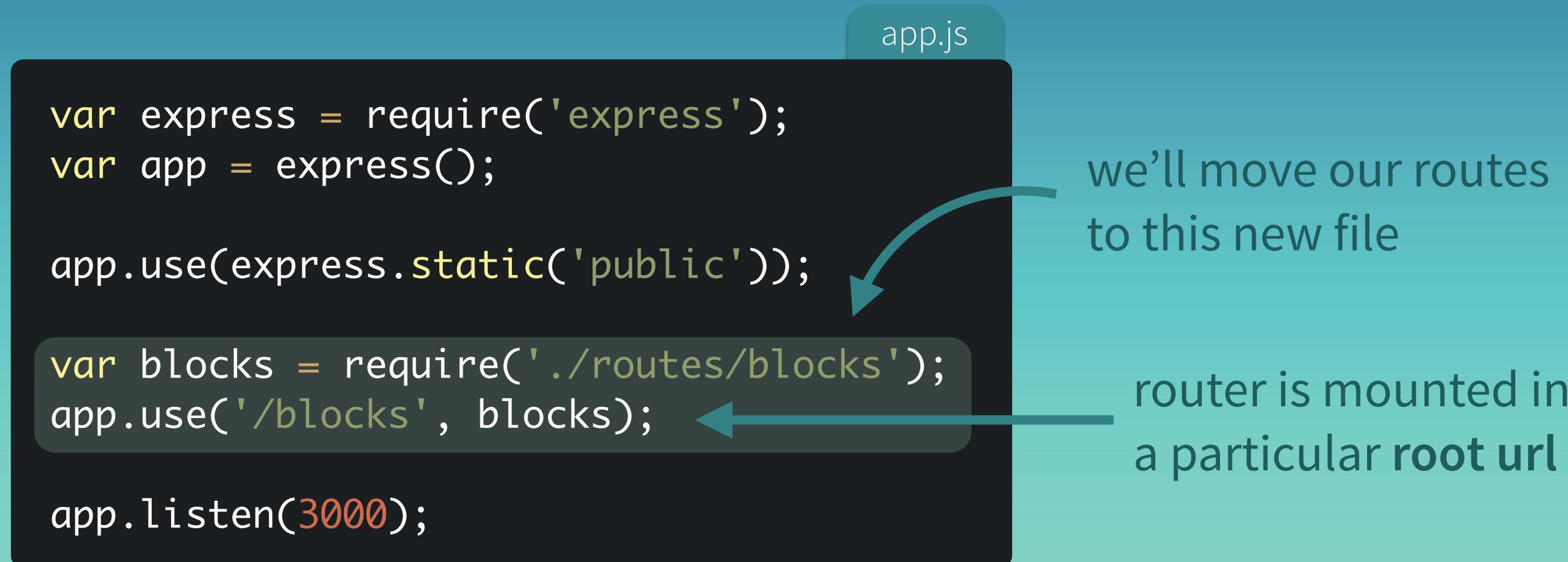
```

Too many lines of code in
a text file is a **bad smell**

Our **app.js** file
is growing too long.

Extracting routes to modules

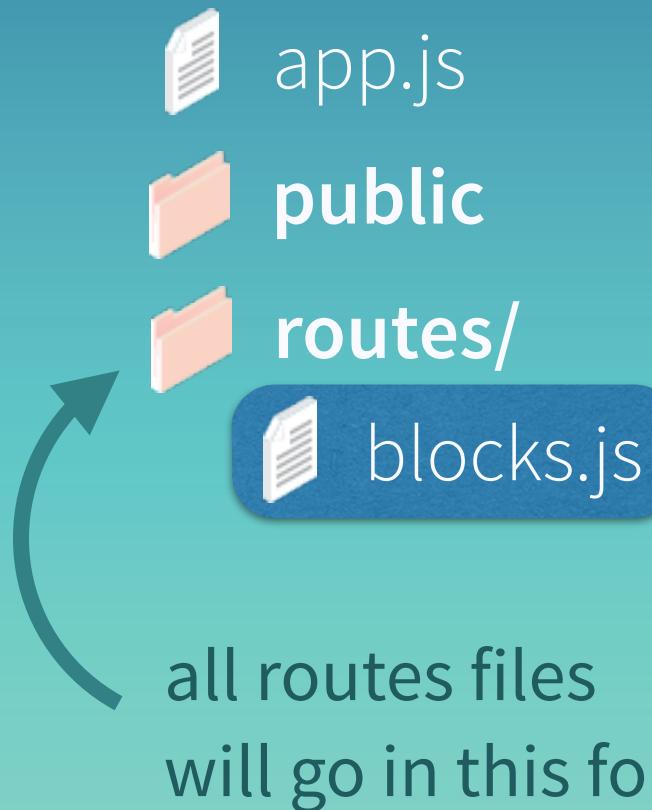
This helps clean up our code and allows our main `app.js` file to easily accommodate additional routes in the future.



Let's see how we can do this by taking advantage of Node's **module** system.

Writing the new routes file

A dedicated folder for routes can help organize our code



```
var express = require('express');
var router = express.Router();
```

blocks.js

returns router instance which can
be *mounted* as a middleware

Exporting the router object

We assign the router to `module.exports` to make it accessible from other files

blocks.js

```
var express = require('express');
var router = express.Router();
```

...

```
module.exports = router;
```

exports the router
as a Node **module**

Extracting GET /blocks to routes file

All block-related logic is encapsulated inside its routes file

blocks.js

```
var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};

module.exports = router;
```

moved here
from app.js

Building the router for /blocks

Router path is **relative** to the where it's mounted

```
var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = {
  'Fixed': 'Fastened securely in position',
  'Movable': 'Capable of being moved',
  'Rotating': 'Moving in a circle around its center'
};

router.route('/')
  .get(function(req, res) {
    res.json(blocks);
  })
  .post(parseUrlencoded, function(req, res) {
    var block = req.body.block;
    if (!blocks[block]) {
      return res.status(400).json({ error: 'Unknown block type' });
    }
    res.json(blocks[block]);
  });

module.exports = router;
```

blocks.js

app.js

```
app.use('/blocks', ...);
```

the **root** path relative
to the **path where it's mounted**

mounted on the
/blocks path

Extracting GET /blocks to routes file

Routes are moved unmodified from the app.js file

blocks.js

```
var express = require('express');
var router = express.Router();
...
router.route('/')
  .get(function (request, response) {
    ...
  });
  .post(parseUrlencoded, function(request, response) {
    ...
  });
...
...
```

same as before

Building the router for /blocks/:name

The route function uses the same format for dynamic routes

```
var express = require('express');
var router = express.Router();
...
router.route('/')
  .get(function (request, response) {
    ...
  });
  .post(parseUrlencoded, function(request, response) {
    ...
  });
router.route('/:name')
  .get(function (request, response) {
    ...
  });
  .post(function (request, response) {
    ...
  });
...

```

blocks.js

app.js

```
app.use('/blocks', ...);
```

the /:name path relative
to the path where it's mounted

mounted on the
/blocks path

Extracting GET /blocks to routes file

The **all** route is called for all requests for a given URL path

blocks.js

```
var express = require('express');
var router = express.Router();
...
router.route('/')
...
router.route('/:name')
  .all(function (request, response, next) {
    var name = request.params.name;
    var block = name[0].toUpperCase() + name.slice(1).toLowerCase();
    request.blockName = block;
    next();
})
...
...
```

the same code that turns first characters to uppercase and remaining characters to lowercase

A diagram illustrating the refactoring process. A curved arrow points from the 'used to be here' placeholder in the original code to the new `app.param` declaration at the bottom right.

```
app.param('name/...'...
```

Extracting GET /blocks to routes file

blocks.js

```
var express = require('express');
var router = express.Router();
...
router.route('/')
...
router.route('/:name')
  .all(function (request, response, next) {
    ...
  })
  .get(function (request, response) {    | same as before
    ...
  });
  .delete(function (request, response) {
    ...
  });
...
...
```



The complete blocks routes file

Our routes file is ready to be mounted in our application

blocks.js

```
var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
var parseUrlencoded = bodyParser.urlencoded({ extended: false });

var blocks = { ... }

router.route('/')
  .get(function (request, response) { ...
  .post(parseUrlencoded, function(request, response) { ...

router.route('/:name')
  .all(function (request, response) { ...
  .get(function (request, response) { ...
  .delete(function (request, response) { ...

module.exports = router;
```

Requiring the router in the application

Our router is simply a Node module and can be required like so



```
app.js
```

```
var express = require('express');
var app = express();

app.use(express.static('public'));

var blocks = require('./routes(blocks');

app.listen(3000);
```

Mounting the router in the application

All requests to the `/blocks` url are dispatched to the `blocks` router



router is mounted in
a particular root url

```
var express = require('express');
var app = express();

app.use(express.static('public'));

var blocks = require('./routes(blocks');
app.use('/blocks', blocks);

app.listen(3000);
```

app.js

Additional route mappings

The app.js file is now ready to support multiple routes and still look clean



app.js

```
var express = require('express');
var app = express();

app.use(express.static('public'));

var blocks = require('./routes/blocks');
var buildings = require('./routes/buildings');
var users = require('./routes/users');

app.use('/blocks', blocks);
app.use('/buildings', buildings);
app.use('/users', users);

app.listen(3000, function () {
  console.log('Listening on 3000 \n');
});
```